

Measuring Complexity by Measuring Structure and Organization

Gregory S. Hornby

Abstract— Necessary for furthering the development of more powerful evolutionary design systems, capable of scaling to evolving more sophisticated and complex artifacts, is the ability to meaningfully and objectively compare these systems by applying complexity measures to the artifacts they evolve. Previously we have proposed measures of modularity, reuse and hierarchy (MR&H), here we compare these measures to ones from the fields of Complexity, Systems Engineering and Computer Programming. In addition, we propose several ways of combining the MR&H measures into a single measure of *structure and organization*. We compare all of these measures empirically as well as on three sample objects and find that the best measures of complexity are two of the proposed measures of structure and organization.

I. INTRODUCTION

Over the years researchers have developed many different evolutionary design systems, some inspired by natural embryogenies and some based on engineering and software development [4], [5]. Necessary to further the science of evolutionary design are metrics that meaningfully compare the complexity of evolved products of these different systems. Ideally, these metrics should give some guidance as to how to construct better evolutionary design systems, such as by measuring those characteristics that are needed for achieving scalability.

As yet, what little work has been done toward developing a theory of complexity and scalability has generally been to propose characteristics, or categories, of representations for evolutionary design systems but not well-defined metrics for measuring them. Angeline classifies representations by whether or not they allow reuse of genotypic elements, and then whether or not the evolved generative system is local to each individual or shared across the population [1]. Bentley and Kumar distinguish between representations which directly encode an object and then distinguish between those that indirectly encode an object implicitly, like cellular automata, or explicitly, like a computer program [3]. Komosinski and Rotaru-Varga list several characteristics of representations for a creature design problem, of which modularity, compression and redundancy are generalizable to other design domains [15]. Stanley and Miikkulainen take five attributes of embryogenies from natural biology as their dimensions for classifying representations – cell fate, targeting, heterochrony, canalization and complexification – but it is difficult to apply these attributes to representations that are not models of developmental biology [22].

One exception in which well-defined metrics are given for an object's description is from our previous work, in which we proposed that modularity, reuse and hierarchy (MR&H)

are the characteristics for improving scalability and in which we defined metrics for measuring them [12]. In this paper we compare our metrics of MR&H against several existing complexity measures from outside the field of Evolutionary Computation. Further, we propose several ways of combining our measures of MR&H into a single measure of the *structure and organization* of an object's encoding.

Our comparison of metrics consists of both an empirical comparison as well as by testing the metrics on three different scenarios. The empirical comparison consists of evolving designs for different sizes of a scalable design problem and is based on the assumption that as the problem scales up, the complexity of evolved designs should increase. The three scenarios consist of examining the complexity scores on hypothetical objects, which are constructed in different ways, and seeing how intuitive they score for them. The results of comparison show that the best measures of complexity are two of our measures of structure and organization.

The rest of this paper is organized as follows. First, we describe our model of design representations, since this is needed to define the metrics operate on them. Second, we present our measures of Modularity, Reuse and Hierarchy (in Section III) and the other measures we compare them against (in Section IV). Next we describe our experimental setup for comparing the different measures in Section V. Then we present the results of comparing the different measures on different sizes of a design problem (Section VI). We follow this with our work in developing a metric for measuring structure and organization by combining the measures of modularity, reuse and hierarchy. Finally we close with a more general discussion of what else needs to be measured and our conclusions.

II. DESIGN ENCODINGS ARE PROGRAMS

Before defining various measures of artifacts, it is worth describing the paradigm under which these measurements are being taken. To define metrics of an artifact in a way that generalizes across various types of artifacts, rather than take measurements of actual fabricated artifacts we take measurements on the data structures that encode these artifacts. These data structures can be thought of as a forest of tree-structured objects, in which each object describes the assembly of some parts, or sub-assemblies, into a larger one (see Figure 1(a)). Since this data-structure defines how the artifact is “built,” we consider it a program for building it. Just as computer programs have procedure calls and iterative loops, so too can design programs have analogous constructs. If we add links to the trees to represent the jumps from these procedure calls and back to the start of an iterative loop this results in this forest of trees really being one large, inter-connected graph

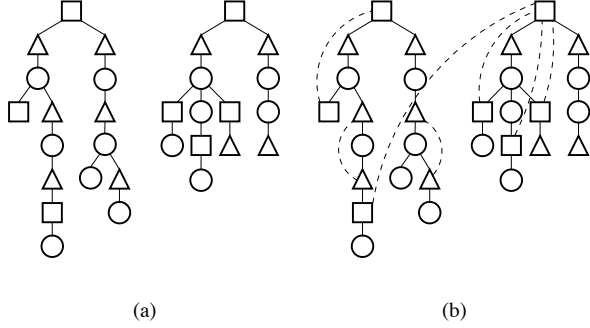


Fig. 1. A graphical rendition of a design program in which different shapes represent different types of operators: (a) the tree structured procedures in the program; (b) the tree-structured procedures in the program with the links added to show procedure calls and the extent of iterative loops.

comprised of multiple sub-graphs for each sub-assembly, and sub-sub-graphs for the sub-sub-assemblies, and so on (see Figure 1(b)). Thus whereas the fields of Computation Theory and Complexity consider programs to be strings (on the tape used by a Turing Machine) here we consider programs to be graphs.

Continuing with the “design encoding is really a design program” metaphor, these graph-structured design programs can be compiled to produce a tree-structured set of design-constructing operators in which all the iterative loops have been unraveled and all the procedure calls have been replaced with the operators inside them. We call this compiled-out tree-structure the assembly procedure for constructing the design.

In the field of Evolutionary Algorithms, the design program is the genotype, the actual artifact (or design) is the phenotype, and the assembly procedure is an intermediate layer between genotype and phenotype. Examples of types of design programs are the artificial genes in a genetic regulatory network, the rules of a cellular automata, and the genotype in Genetic Programming. In the experiments in this paper the genotype we are evolving is the design program, and it is compiled into an assembly procedure which is then sent to a design-constructor to create the phenotype – this will be described in more detail in Section V-B. Some of the measures presented in the following two sections will be of the design program and some of them be of the assembly procedure.

III. MEASURING MODULARITY, REUSE AND HIERARCHY

The method for measuring MR&H comes out of what is meant by these terms. *Modularity* is defined as an encapsulated group of elements that can be manipulated as a unit, *reuse* is a repetition or similarity in a design, and *hierarchy* is the number of layers of encapsulated modules in the structure of a design.

Each of MR&H aids the scalability of evolutionary design systems in different ways. Modularity is related to the building block hypothesis of genetic algorithms (GAs) [9], which

states that GAs work by testing groups of basic components and combining them to form highly fit solutions. Simon, in his parable of two watchmakers, illustrated how the ability to create and manipulate modules greatly improves the rate at which more structurally sophisticated artifacts can be built [20]. For larger and more sophisticated artifacts, being able to hierarchically create levels of nested modules is needed to break things down so no one module is too large and sophisticated to evolve on its own. Being able to reuse design modules is helpful in two ways. First, a module that is useful in one part of the design may be useful somewhere else so creating modules is a way of scaling the basic unit of variation. Second, reuse of a parameter, assembly or function is a way of capturing design dependencies into a single location in the design encoding thereby enabling variation operators to more easily make coordinated changes in the design. The measures of MR&H can now be defined.

Modularity: The modularity value of a design is a count of the number of structural modules in it, which we define as an encapsulated group of elements in the design encoding that can be manipulated as a unit. Since a label to a procedure can be manipulated as a unit, each procedure in the design encoding counts as one toward the encoded modularity value. In addition, the ability to change the iteration counter means that the group of encoded elements inside an iterative block also constitute a module, hence each iterative block is one module in the encoding. As well as counting modules in the encoded design (which we label M_p , for modules in the program) we can also count the number of occurrences of modules in the design itself, M_d . In this case each procedure call counts as one toward the design modularity value and each iteration of an iterative block adds one to the modularity value of the design.

Reuse: is a measure of the amount of reuse of genotypic elements in creating the phenotype. It can be calculated by counting the “size” of the object and dividing this by the size of the encoding. For example, the amount of reuse in a string is the size of the string divided by the size of the program that generates the string. Here we measure three types of reuse. The first, overall reuse, R_a , is the average amount of reuse of a symbol and is calculated as the size of the design’s assembly procedure divided by the size of the design program. Second, reuse of build symbols, R_b , is the average number of times a design constructing operator – as opposed to an operator that is a conditional, iterative statement or procedure call – is used. Third, reuse of modules, R_m , is the average number of times modules are reused in the design and is calculated as M_d divided by M_p .

Hierarchy: The hierarchy of an encoded design is a measure of the number of nested layers of modules, such as through iteration or abstraction. A design encoding with no modules has a hierarchy of zero. Each nested module, whether a successful call to a labeled procedure or a non-empty iterative block, increases the hierarchy value by one. This is similar to measuring the depth of an object’s assembly sequence [8], but whereas there the measure is of basic steps

in constructing an object, here we are measuring modules of basic steps.

As defined, these measures of MR&H apply to any programming language, and are thus comparable on the same systems as existing complexity measures, such as AIC, Logical Depth and Sophistication. These measures can also be generalized to any representation with a hierarchical graph structure, such as the set of parts used to describe a complex assembly in a CAD/CAM package, and any system that can be described as a hierarchical graph structure, such as a regular expression. Not as obvious is how to apply these measures to non-procedural representations such as DNA and artificial genetic regulatory networks, for which the challenge is mainly the identification of modules.

In the rest of this paper we use MRH to refer to the metrics for modularity, reuse and hierarchy and MR&H to refer to the characteristics of modularity, reuse and hierarchy. Also, in Section VII we discuss how to combine these measures of MRH into a single measure, which we call a measure of *structure and organization*.

IV. COMPLEXITY METRICS

Even though the MRH metrics were developed specifically for computer-automated design they may not be any better for comparing the sophistication of evolved designs as similar metrics developed in related fields. One of the objectives of this paper is to compare these metrics against ones that already exist. For this comparison we selected those metrics which are relatively straightforward to compute or approximate and which we thought had a reasonable chance at being relevant. Example of some of measures we left out are: Arithmetic Complexity, Cognitive Complexity, Dimension of Attractor, Ease of Decomposition, Logical Complexity, Mutual Information, Number of Inequivalent Descriptions, Number of States in a Finite Automata, Number of Variables, Thermodynamic Depth, and Variety. (all of these measures, as well as several others, are reviewed in [7]). We now present the metrics which we compare MRH against.

Algorithmic Information Content (AIC) is one of most well known and influential complexity metrics, having been used as a starting point for many others, and was invented separately by Chaitin [6], Kolmogorov [14], and Solomonoff [21]. The AIC of a given string is the length, in number of symbols, of the shortest program that produces that string. For this work we estimate the AIC by calculating the number of symbols in the design program since this is the evolved genotype that defines the object. While it is likely that some of the evolved genotypes could be compressed, using their size is a simple upper bound on AIC and is a correct measure of the size of the program that was evolved. This measure is also analogous to counting the number of lines in a computer program, which is one measure of its complexity [7].

Design size is a measure of the size of what is encoded by the design program, and here we measure this by counting the number of symbols in the assembly procedure. In the field of Complexity, in which there is a string and program

that produces that string, this measure would be a count of the size of the string.

Logical Depth is a measure of the value of information and, for a given string, it is the minimum running time of a near-incompressible program that produces the string [2]. In this case we use the evolved design program as the near-incompressible program and calculate the running time of this program as the number of symbols that are processed in generating the assembly procedure. This can also be considered computation complexity, in that it is a measure of the amount of computational time that is spent to compute the assembly procedure.

Sophistication is a measure of the structure of a string by counting the number of *control* symbols in the design program used to generate it [16]. In trying to measure the structure of a string, the goal for this measure is similar to the goal of MRH metrics. Here we calculate the sophistication of a design by counting the number of control symbols – that is, procedure symbols, loop symbols, conditionals – in the program that is used to generate it.

Number of Build Symbols, whereas Sophistication is a measure of structure by counting the number of control symbols, we propose a counter measure which is a count of the number of non-control symbols in the design program that is used to generate the assembly procedure. In our system, these non-control symbols are the operators that are used by the design-constructing interpreter and we call them *build* symbols, since they are used to build a 3D shape.

Grammar Size: any string that has a pattern can be expressed as being generated by a grammar. Simple strings, with simple patterns, generally have a simple grammar thus the size of the grammar needed to produce a string serves as a measure of complexity [7]. Since the representation used here is based on parametric Lindenmayer systems [18], (although it is more like Genetic Programming [17]) the procedures can be thought of as grammar rules. To calculate the grammar size of an assembly procedure we use the design program that produces it as the grammar and count the number of production-rules in it.

Connectivity: more complex systems have greater inter-connectedness between components, thus the connectivity of a system can be used as a complexity measure [7]. For a graph-structure, its connectivity is the maximum number of edges that can be removed before it is split into two non-connected graphs. To calculate the connectivity of a design we use the connectivity of the design program that is used to generate it, since this program has a graph-structure to it.

Number of Branches: inspired by the previous measure of complexity, another measure of the structure of a graph is a count of number of nodes which are branch nodes – nodes which have two or more children. Strings have a very simple structure with no branching nodes, whereas a fully balanced binary tree will have roughly $\lg(n)$ branch nodes. For this measure we count the number of branches in the assembly procedure produced by the design program.

Height: is the maximum number of edges that can be

traversed in going from the root of the tree to a leaf node. Unlike other complexity metrics, which are based on strings, this measure is for trees and here we apply this measure to the assembly procedure that is generated by the design program. This measure of complexity is related to work in formal language theory in which ideas for measuring ease of comprehension are to measure the depth of postponed symbols [23] or depth and nesting, called Syntactic Depth, [19].

V. EXPERIMENTAL SETUP

To compare metrics we use an existing evolutionary design system on a previously used benchmark problem. The evolutionary design system we use is GENRE, and this system allows the user to select which of combinations of MR&H to enable by selecting which aspects of a programming language (conditionals, labeled procedures, and iterative loops) are available to the representation [11]. To compare the different metrics of complexity we perform runs with them on different sizes of a scalable design problem.

A. Test Problem

For the experiments in this paper the design problem we use is that of producing a 3D table out of cubes, for which the fitness function for scoring tables is a function of their height, surface structure, stability and number of excess cubes used [10]. Height is the number of cubes above the ground. Surface structure is the number of cubes at the maximum height. Stability is a function of the volume of the table and is calculated by summing the area at each layer of the table. Maximizing height, surface structure and stability typically results in table designs that are solid volumes, thus a measure of excess cubes is used to reward designs that use fewer bricks,

$$\begin{aligned}
 f_{height} &= \text{the height of the highest cube, } Y_{max}. \\
 f_{surface} &= \text{the number of cubes at } Y_{max}. \\
 f_{stability} &= \sum_{y=0}^{Y_{max}} f_{area}(y) \\
 f_{area}(y) &= \text{area in the convex hull at height } y. \\
 f_{excess} &= \text{number of cubes not on the surface.}
 \end{aligned}$$

To produce a single fitness score for a design these five criteria are combined together:

$$\text{fitness} = f_{height} \times f_{surface} \times f_{stability} / f_{excess} \quad (1)$$

This problem can be scaled by varying the size of the grid. In our experiments we do runs with sizes from $20 \times 20 \times 20$ to $80 \times 80 \times 80$.

The design constructor for making table designs starts with a single cube in an otherwise empty 3D grid and then executes the assembly procedure that was produced from compiling the genotype. Cubes are added to this design with the operators `forward()` and `backward()`. The current state, consisting of location and orientation, is maintained with the addition of cubes resulting in a change in the current

location and the `rotate-xyz()` operators change the current orientation. A branching in the assembly procedure results in a split in the construction process with construction continuing with each child subtree working with its own copy of the construction state.

B. Representation

In the following example we demonstrate the representation and method for creating a design as well as calculate its complexity scores using the different complexity metrics. This example design encoding consists of a starting command, `Proc.0(4.0,2.0)`, and two labeled procedures, `Proc.0` and `Proc.1`, each with two parameters:

`Proc.0(4.0,2.0) :`

`Proc.0(n_0, n_1) :`

```

 $n_0 > 3.0 \rightarrow$  rotate-z(1) [ Proc.0(1.0,2.0) repeat(2) [ forward( $n_1/2$ ) [ repeat-end
[ Proc.1( $n_0+2.0,2.0$ ) [ forward(1) ]
] [ ] [ ] ] ]
true  $\rightarrow$  rotate-z(1) [ repeat(4) [ rotate-
y(1) [ forward( $n_1+1.0$ ) repeat-end [
rotate-x(1) ] ] ] [ ] ]

```

`Proc.1(n_0, n_1) :`

```

 $n_0 > 1.0 \rightarrow$  forward(2) [ Proc.1(1.0, $n_1+1.0$ )
[ forward(1) ] rotate-y(2) [ [ ]
Proc.1(1.0, $n_1+1.0$ ) [ forward(1) ] ]
Proc.1( $n_0-2.0, n_1-1.0$ ) [ end-proc ]
]
 $n_0 > 0.0 \rightarrow$  rotate-y(1) [ [ ] backward( $n_1$ ) [ end-
proc [ ] ] ]

```

A graphical version of this design program is shown in Figure 1.

To generate the assembly procedure for this design program it is executed, starting with the statement `Proc.0(4.0,2.0)`. This results in the following assembly procedure:

```

rotate-z(1) [ rotate-y(1)
[ forward(3) rotate-y(1) [ forward(3)
rotate-y(1) [ forward(3) rotate-y(1)
[ forward(3) rotate-x(1) ] ] ] [ ] ]
forward(1) [ forward(1) [ forward(2) [
rotate-y(1) [ [ ] backward(3) [ forward(1)
[ ] ] ] rotate-y(2) [ [ ] rotate-y(1) [
[ ] backward(3) [ forward(1) [ ] ] ] ]
forward(2) [ rotate-y(1) [ [ ] backward(2)
[ forward(1) [ ] ] ] rotate-y(2) [ [ ]
rotate-y(1) [ [ ] backward(2) [ forward(1)
[ ] ] ] ] forward(2) [ rotate-y(1) [ [ ]
backward(1) [ forward(1) [ ] ] ] rotate-
y(2) [ [ ] rotate-y(1) [ [ ] backward(1) [
forward(1) [ ] ] ] ] forward(1) [ ] ] [ ]
[ ] ] [ ] [ ] ] ]

```

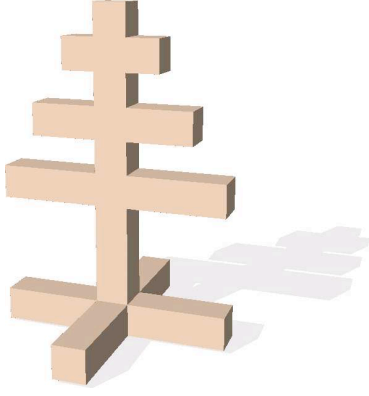


Fig. 2. The 3D object that is constructed from the example design encoding.

This example design can be analyzed using the metrics of MRH and the various complexity measures. The program has six modules that are used a total of 17 times giving a modularity value of 6 for the encoding and a modularity value of 17 for the design. The size of the program is 30 symbols and the size of the final assembly procedure is 38 symbols giving a reuse value of 1.27, and it has five levels of nested modules which gives it a hierarchy value of 5. Its scores on the other complexity measures are: an AIC of 30; a Design size of 38; a Logical Depth of 124; a Sophistication of 21; 13 build symbols; a grammar size of 2; a connectivity of 5; 8 branches; and a height of 10.

C. Evolutionary Algorithm

The EA used for the experiments is the Age-Layered Population Structure (ALPS) [13]. Unlike a traditional EA, ALPS maintains several layers of individuals of different age levels and continuously introduces new, randomly generated individuals into the first layer. It has been shown to work better than the canonical EA by better avoiding premature convergence. The setup we use consists of 10 layers, each with 40 individuals. In each layer the best 2 individuals from the previous generation are copied to the current generation and then new individuals are created with a 40% chance of mutation and 60% chance of recombination. Tournament selection with a tournament size of 5 is used to select parents. In our experiments we run 15 trials with each configuration and each trial is run for one million evaluations.

VI. EMPIRICAL COMPARISON

To compare complexity and MRH metrics we ran a number of experiments on different sizes of a design problem. The design problem and evolutionary algorithm were described in the previous section, and for these experiments we performed four sets of experiments in which we evolved tables for four different grid sizes. Here we are working with the assumption that a more “complex” design is needed to produce good designs for a larger design space and so we are looking for complexity metrics whose values scale similarly to design size.

Figure 3 contains images of two of the best and most structurally organized tables that were evolved. The smaller

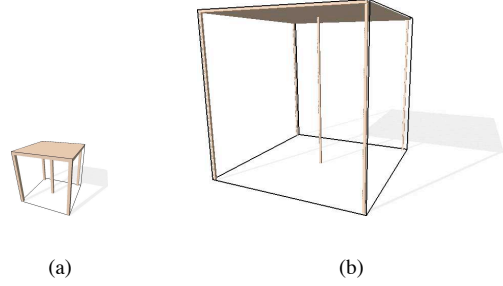


Fig. 3. Two of the best, and most structurally organized, of the evolved tables. The first (a) was evolved in the $20 \times 20 \times 20$ design space and the second (b) was evolved in the $80 \times 80 \times 80$ design space.

TABLE I

A COMPARISON OF THE RESULTING SCORES ON THE DIFFERENT METRICS OF THE BEST TABLES EVOLVED WITH THE DIFFERENT REPRESENTATIONS. RESULTS ARE THE AVERAGE OVER 15 TRIALS.

	20^3	40^3	60^3	80^3
Fitness ($\times 10^6$)	0.56	18.1	123	440
AIC	719	768	680	775
Design Size	6769	9499	9739	9944
Log. Depth	9541	13421	14376	18011
Sophistication	79.9	70.53	74.0	85.4
# Bld Sym	626	684	593	676
Grammar Size	13.5	13.2	12.5	13.5
Connectivity	33.7	25.2	26.4	37.3
# Branches	1653	2087	1905	1825
Height	118	145	276	220
Modularity (M_p)	27.5	26.1	30.8	31.1
Mod. in Design (M_d)	377	547	1133	1329
Reuse (R_a)	12.1	14.0	16.6	15.7
Reuse of Bld. (R_b)	15.2	16.2	19.6	18.5
Reuse of Mod. (R_m)	15.2	21.8	37.4	50.1
Hierarchy (H)	7.53	7.7	8.0	8.6

table, Figure 3(a), was evolved in the $20 \times 20 \times 20$ design space and has a fitness of 582221 and the following scores: AIC of 913; Design Size of 8007; Logical Depth of 10311; Sophistication of 89; 811 build symbols; a Grammar Size of 13; a Connectivity of 34; 1595 branches; and a height of 155. Its MRH scores are: M_p is 34, M_d is 431; R_a is 8.8; R_b is 9.9; R_m is 12.7 and it has an H of 8. The larger table, Figure 3(b), was evolved in the $80 \times 80 \times 80$ design space and has a fitness of 600324286 and the following scores: AIC of 630; Design Size of 9753; Logical Depth of 14365; Sophistication of 90; 529 build symbols; a Grammar Size of 11; a Connectivity of 58; 1668 branches; and a height of 168. Its MRH scores are: M_p is 20, M_d is 2202; R_a is 15.5; R_b is 18.4; R_m is 110.1 and it has an H of 9. While these scores give examples of the differences that can happen, a better overall picture is gained from looking at the average scores from a number of evolutionary runs on different sizes of the design problem.

Table I lists the average values over 15 trials of the different measures as applied to the best tables evolved on different sizes of the design problem ($20 \times 20 \times 20$, $40 \times 40 \times 40$, $60 \times 60 \times 60$, and $80 \times 80 \times 80$). As expected, the averaged best fitness monotonically increases along with

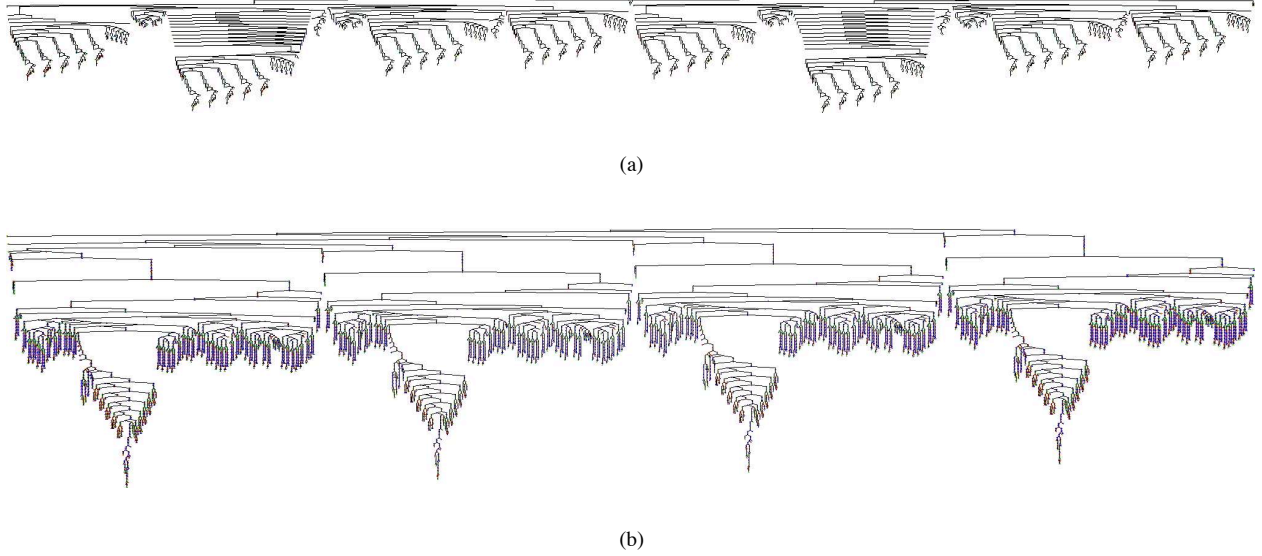


Fig. 4. A graphical rendition of the assembly procedures for constructing the two tables in Figure 3.

an increase in size of the design space. The measures which have values that also monotonically increase in step with an increase in size of the design space are: Design Size, Logical Depth, M_d , R_m , and H . Of these it is not surprising that Design Size increases with the size of the design space and, given that the Design Size increases, it is also not surprising that Logical Depth (a measure of the running time of the program that creates the assembly procedure) also increases with size of the design space. Interestingly, the information in a design, AIC, does not grow monotonically with size of the design space or Design Size. In addition, none of the other measures grows monotonically with the size of the design space except some of the measures of structure and organization: the amount of modularity in the design (M_d), the reuse of modules (R_m) and hierarchy (H).

Of the three measures of reuse, R_a , R_b and R_m , only modular reuse (R_m) monotonically increases with the size of the design space and the fitness of the best designs. This suggests that the type of reuse that is useful is not overall reuse (R_a) or reuse of build symbols (R_b), but the reuse of modules. By extension, this also suggests that those design representations which do not have the ability to hierarchically assemble and reuse modules will not scale well.

Of the two modularity measures M_d monotonically increased along with the increase in fitness and size of the design space whereas M_p was higher in the $20 \times 20 \times 20$ space than in the $40 \times 40 \times 40$ space. Since M_d is a product of the number of modules in a design program (M) and the amount of reuse of these modules (R_m) it may be a more reliable measure of “complexity” because it is a combination of two separate aspects: modularity and a modular reuse. This suggests that measuring modularity alone is not a good overall measure of the complexity of an object and that combining the measures of all three characteristics of MR&H

into a single measure may result in an even better measure of an object’s structure and organization.

VII. MEASURES OF STRUCTURE AND ORGANIZATION

Each of the proposed metrics of modularity, reuse and hierarchy measure different aspects of the structure and organization of an object. Of interest is combining the scores of these three metrics into a measure of structure and organization with a single value, for which there are various methods of doing this. One method for combining the three scores of MRH into a single value is by treating each of them as the orthogonal axes of a 3D system and then using the length of the vector from the origin as the measure of structure and organization of an object. Since the measure of modular reuse worked well on its own, we also include a variant of this measure of structure and organization using modular reuse instead of overall reuse.

$$SO_1 = \sqrt{M^2 + R_a^2 + H^2} \quad (2)$$

$$SO_2 = \sqrt{M^2 + R_m^2 + H^2} \quad (3)$$

A problem with this approach is that the different metrics vary in their range, and a small change in hierarchy will generally have little impact on the overall structure and organization measure of an object since hierarchy usually has the smallest value.

Another method for combining the three MRH scores is to simply multiply them together.

$$SO_3 = M \times R_a \times H \quad (4)$$

$$SO_4 = M \times R_m \times H \quad (5)$$

This approach has the desirable property that a change of $X\%$ in any one of MRH will result in the same $X\%$ change in the overall measure of structure and organization,

TABLE II

DIFFERENT WAYS OF COMBINING MRH SCORES TO PRODUCE A SINGLE MEASURE OF STRUCTURE AND ORGANIZATION.

		20 ³	40 ³	60 ³	80 ³
	Fitness ($\times 10^6$)	0.56	18.1	123	440
SO_1 :	$\overrightarrow{MR_a H}$	31.3	31.1	37.1	37.1
SO_2 :	$\overrightarrow{MR_m H}$	34.0	36.0	51.6	64.4
SO_3 :	$M \times R_a \times H$	2013	2872	3708	4019
SO_4 :	$M \times R_m \times H$	2889	4324	8643	11207
SO_5 :	$\frac{M \times R_a \times H}{AssemSize}$	0.31	0.31	0.38	0.40
SO_6 :	$\frac{M \times R_m \times H}{AssemSize}$	0.42	0.46	0.89	1.13
SO_7 :	$M \times R_a \times H / AIC$	3.22	4.68	6.75	6.77
SO_8 :	$M \times R_m \times H / AIC$	4.59	6.87	15.4	19.3

Of concern with the initial approaches to measuring structure and organization are that they do not take into account either the size of the object or the amount of information in it. For example, a large object with a small percentage of its information organized into some structure can outscore a much smaller object which has a small, maximally-organized, design program. Two ways to normalize structure and organization scores for size are to divide by the size of the object and to divide by the amount of information in the object.

$$SO_5 = \frac{M \times R_a \times H}{DesignSize} \quad (6)$$

$$SO_6 = \frac{M \times R_m \times H}{DesignSize} \quad (7)$$

$$SO_7 = \frac{M \times R_a \times H}{AIC} \quad (8)$$

$$SO_8 = \frac{M \times R_m \times H}{AIC} \quad (9)$$

Table VII contains the scores for these different measures of structure and organization (SO) on the best design programs evolved for different sizes of the design problem. Of these eight measures of structure and organization, neither SO_1 and SO_5 , both of which use overall reuse and not modular reuse, increase monotonically along with the size of the design space. The other six measures of structure and organization do increase monotonically, with the four measures of structure and organization which use modular reuse (R_m), instead of overall reuse, scaling in a way that better matches the increase in design space and the increase in fitness.

VIII. COMPARING MEASURES ON EXAMPLES

One shortcoming with some measures of complexity, such as AIC, is that they are not very intuitive. We can examine how intuitive these measures of structure and organization are by trying them on a couple of examples. First, consider the AIC of an algorithmically random bit string, by which is meant one with no regularities. Since the string has no regularities it cannot be compressed, so its AIC is the size of the string plus the overhead necessary for the `print` operator. Compare this to the MRH and structure and organization values of this string: its modularity value is 0, since

it has no modules, its reuse value is 1, since there are no reused symbols, and its hierarchy value is 0, since there is no modules to be nested. Using these values, its various structure and organization values ($SO_1 \dots SO_8$) are: 1, 1, 0, 0, 0, 0, 0, and 0. These values of 0 and 1 for the measures of MRH and structure and organization match our intuition that a random string does not have a sophisticated structure.

Next, consider what happens to the structure and organization values when an object, A_1 , is joined to itself to form a new object, A_2 . In this case the design program of the new object, A_2 , would be the same as for the original object, plus the module, $A_2 = A_1 + A_1$. As a result of this new module, the hierarchy of A_2 would be $H(A_1)$ plus 1 and the modularity would be $M_p(A_1)$ plus 1. Depending on the AIC of A_1 , the amount of reuse will be up to a factor of 2 larger for the new object since $R(A_2) = \frac{DS(A_1) + DS(A_1) + k}{AIC(A_1)}$, where k is the size of adding the new module and $DS(A)$ is the Design Size of A . As a result of these changes in MRH, the structure and organization values of SO_5 through SO_8 should be only slightly larger, but those of $SO_3(A_2)$ and $SO_4(A_2)$ will be roughly double that of A_1 . Consider what happens to other scores of complexity: AIC, Sophistication and Grammar Size increase slightly but Logical Depth doubles. Since A_2 is just two copies of A_1 , it is not clear that it should have twice the complexity of A_1 , thus measures SO_5 through SO_8 are more intuitive than $SO_3(A_2)$, $SO_4(A_2)$ and logical depth on this example.

Similarly, consider the case in which two completely different objects, A_1 and A_3 , with the same complexity and MRH scores, are combined to form a new object, A_4 : $A_4 = A_1 + A_3$. In this case the new module results in the hierarchy of the new object being one plus the hierarchy of either of its component objects: $H(A_4) = H(A_1) + 1 = H(A_3) + 1$. The modularity of this new object is equal to one plus the sum of its to component objects: $M(A_4) = M_p(A_1) + M_p(A_3) + 1$. Whereas both modularity and hierarchy increase, this new object has a reuse slightly less than both of its component objects since the size of the phenotype is $DS(O_1) + DS(O_3)$ but the size of the genotype is $AIC(O_1) + AIC(O_3)$ plus some additional symbols for specifying $A_4 = A_1 + A_3$.¹ Thus SO_3 and SO_4 would be (roughly) double in value for A_4 as they are for A_1 and A_2 , but SO_5 through SO_8 would change little since both AIC and design size would also (roughly) double in size. Not only would AIC for A_4 be roughly double that of either A_1 or A_3 , but so would Logical Depth, Sophistication, and Grammar Size. Just as combining an object with itself does not seem like it should lead to a doubling in complexity, neither does it seem that combining two completely different objects with the same complexity should lead to a doubling of complexity. Thus, as with the previous example, we find that the more intuitive measures are SO_5 through SO_8 .

¹To be precise, the design programs for both A_1 and A_3 have a starting rule, one of these is kept and is changed to call the new rule, $A_4 = A_1 + A_3$, and the other starting rule is deleted so the AIC of A_4 is only a couple of symbols larger than $AIC(A_1) + AIC(A_3)$.

To summarize the results of these three examples we can state some desirable properties of a measure of complexity:

- 1: The complexity value of a random string should be small.
- 2: The complexity value of an object joined to itself should be only slightly larger than that of the original object.
- 3: The complexity value of two objects joined together should not be smaller than the lesser value of the two original objects and should not be much larger than the greater value of the two original objects.

Using these principles, and the results of the experiments in Section VI, the best measures of complexity are SO_6 and SO_8 .

IX. CONCLUSION

Necessary for the advancement of scalable evolutionary systems is the identification of the fundamental properties of such systems and metrics for measuring them. As yet, the only clear metrics to come from the field of Evolutionary Computation are measures of modularity, reuse and hierarchy (MRH). Here we compared the MRH measures to various measures of complexity from other fields with the goal of identifying which ones best scale with what we intuitively think of as complexity. In addition, we proposed various measures of *structure and organization* by combining the measures of MRH in various ways.

Working with the hypothesis that by scaling the size of a problem, more “complex” solutions are required to solve it, we compared all measures both empirically, as well as against the other complexity measures on measure three example objects. All of the other complexity measures either failed to scale correctly in our empirical comparison, or gave unintuitive results for at least one of our example objects. Of the measures we proposed, two of our measures of structure and organization passed all of the tests and so we conclude that the best measures of complexity are SO_6 and SO_8 , which are the product of multiplying the MR&H measures together, and then normalizing by either dividing by AIC (SO_6) or by dividing by the design size (SO_8).

To summarize, while the amount of information in a design is certainly an important factor for measuring complexity, AIC produces unintuitive scores for various examples. Similarly, the other measures of complexity (Logical Depth, Sophistication, ...) all fail to pass at least one of our three proposed properties of a complexity measure. That two measures of structure and organization pass the empirical experiments and also correspond well with what we intuitively think of as complex indicates that it is not so much the amount of information that is important in determining complexity, rather it is how this information is structured and organized.

REFERENCES

- [1] P. J. Angeline. Morphogenic evolutionary computations: Introduction, issues and examples. In J. McDonnell, B. Reynolds, and D. Fogel, editors, *Proc. of the Fourth Annual Conf. on Evolutionary Programming*, pages 387–401. MIT Press, 1995.
- [2] C. H. Bennett. On the nature and origin of complexity in discrete, homogenous, locally-interacting systems. *Foundations of Physics*, 16:585–592, 1986.
- [3] P. Bentley and S. Kumar. Three ways to grow designs: A comparison of embryogenies of an evolutionary design problem. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Genetic and Evolutionary Computation Conference*, pages 35–43. Morgan Kaufmann, 1999.
- [4] P. J. Bentley, editor. *Evolutionary Design by Computers*. Morgan Kaufmann, San Francisco, 1999.
- [5] P. J. Bentley and D. W. Corne, editors. *Creative Evolutionary Systems*. Morgan Kaufmann, San Francisco, 2001.
- [6] G. J. Chaitin. On the length of programs for computing finite binary sequences. *Journal of the Association of Computing Machinery*, 13:547–569, 1966.
- [7] B. Edmunds. *Syntactic Measures of Complexity*. PhD thesis, Dept. of Philosophy, University of Manchester, 1999.
- [8] M. Goldwasser, J. Latombe, and R. Motwani. Complexity measures for assembly sequences. In *Proc. IEEE Intl. Conf. on Robotics and Automation*, pages 1581–1587, Minneapolis, MN, Apr. 1996.
- [9] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, 1975.
- [10] G. Hornby. Functional scalability through generative representations: the evolution of table designs. *Environment and Planning B: Planning and Design*, 31(4):569–587, July 2004.
- [11] G. S. Hornby. *Generative Representations for Evolutionary Design Automation*. PhD thesis, MIT School of Computer Science, Brandeis University, Waltham, MA, 2003.
- [12] G. S. Hornby. Measuring, enabling and comparing modularity, regularity and hierarchy in evolutionary design. In H.-G. B. et al., editor, *Proc. of the Genetic and Evolutionary Computation Conference, GECCO-2005*, pages 1729–1736, New York, NY, 2005. ACM Press.
- [13] G. S. Hornby. ALPS: The age-layered population structure for reducing the problem of premature convergence. In M. K. et al., editor, *Proc. of the Genetic and Evolutionary Computation Conference, GECCO-2006*, pages 815–822, New York, NY, 2006. ACM Press.
- [14] A. N. Kolmogorov. Three approaches to the quantitative definition of information. *Problems of Information Transmission*, 1:1–17, 1965.
- [15] M. Komosinski and A. Rotaru-Varga. Comparison of different genotype encodings for simulated 3d agents. *Artificial Life*, 7(4):395–418, 2001.
- [16] M. Koppel. Complexity, depth and sophistication. *Complex Systems*, 1:1087–1091, 1987.
- [17] J. R. Koza. *Genetic Programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, 1992.
- [18] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.
- [19] B. K. Rosen. Syntactic complexity. *Information and Control*, 24:305–335, 1974.
- [20] H. A. Simon. *The Sciences of the Artificial*. MIT Press, Cambridge, MA, 1969.
- [21] R. J. Solomonoff. A formal theory of inductive inference. *Information and Control*, 7:1–22, 224–254, 1964.
- [22] K. O. Stanley and R. Miikkulainen. A taxonomy for artificial embryogeny. *Artificial Life*, 9(2):93–130, 2003.
- [23] V. H. Yngve. A model and an hypothesis for language structure. In *Proceedings of the American Philosophical Society*, pages 444–466, 1960.